M2104 Conception Objets – Notes de cours – Rendre compte d'un programme existant
Session 1

1 Cours : conception et modélisation par objets

1.1 Les concepts

Conception et modélisation sont confrontées depuis fort longtemps : la tradition philosophique européenne fait remonter à Platon la vision d'*idées* qui serviraient de moule à tout ce qui est alors que son disciple Aristote les concevait comme des *catégories* créées par l'esprit humain qui cherche les éléments similaires dans son environnement. De la même façon, on peut envisager la préparation d'un programme par objets d'un côté comme une activité *créatrice* dans laquelle le programmeur est l'auteur : les classes qu'il écrit seront les modèles des objets, leurs instances. On parle alors *conception*. On peut aussi envisager cette préparation d'un programme par objets comme une activité *analytique*, dans laquelle le programmeur est un analyste qui cherche à extraire les comportements d'un phénomène physique, d'un processus dans une entreprise, ou du fonctionnement d'un monde fictif qu'un scénariste de jeux vidéo vient de lui décrire : le programmeur cherche comment représenter avec des classes les éléments caractérisant les différents objets qu'il rencontre. On parle ici de *modélisation*. Conception et modélisation vont en fait souvent de pair, et le dialogue entre ces deux approches permet de converger vers une spécification complète d'un modèle par objets et classes du programme à réaliser ou en cours de réalisation.

1.2 Objectifs du cours

Pour ce cours de conception et modélisation par objets, le but est d'apprendre, d'une part, à rendre compte de ses choix, et, d'autre part, à préparer des spécifications complètes servant de socle à des projets informatiques.

1.3 Moyens et évaluation

Le cours se déroulera uniquement en séances de TD intégrant cours et exercices. Il sera évalué au moyen d'une note de TD comportant surtout l'évaluation d'un travail à la maison (à réaliser en binôme, coefficient 3), d'une interrogation écrite (DS court, coefficient 2) et d'un DS terminal (coefficient 5). Le module M2104 lui-même est de coefficient 2,5 sur le semestre 2.

1.4 Les objets, les classes

Dans le paradigme de programmation par objets (orientée objets, ou simplement objet), les *objets* désignent des unités dans les programmes qui :

- fournissent des services à d'autres objets (ce sont les *méthodes* : on parle d'opérations, de traitements ou de comportements, *ce que fait l'objet* ou *ce à quoi il sert*),
- contiennent les données sur lesquelles ils opèrent (ce sont les *attributs*).

Les objets sont définis par des *classes* (en Java, il est nécessaire de disposer d'une classe – explicite ou non – pour créer un objet, ce n'est pas le cas de tous les langages). Un objet est une *instance* d'une classe. La relation entre classes et objets est celle entre types et individus (les entiers naturels sont un type, 7 est un individu de ce type ; `String` est une classe Java, dans `String s = "Bonjour"; s` est un objet de cette classe). La programmation objet en Java consiste à définir dans les classes le comportement qu'auront les objets de cette classe (leurs *méthodes d'instance*), les données sur lesquelles ils s'appuient (leurs *attributs d'instance*), ainsi que des opérations et données partagées au sein de la classe (les méthodes et attributs *de classe*). Il s'agit, dans un premier temps, de rendre compte de l'*organisation* de ces classes : quelles sont leurs traitements, leurs données, et les relations entre elles.

1.5 Le langage

Pour rendre compte des choix effectués et communiquer efficacement entre membres et entre équipes de conception, développement, programmation, test, maintenance... mais aussi avec les clients et les prestataires extérieurs, un langage clair doit être utilisé. UML, *the Unified Modelling Language*, est celui étudié ici ; c'est l'un des grands langages de modélisation de l'industrie, basé sur de multiples diagrammes permettant de spécifier de nombreux aspects de logiciels, de systèmes d'information, ou de processus métier. Notre objectif premier est de nous accorder sur les bases de certains diagrammes UML ; avant de pouvoir vous approprier ce langage et de l'utiliser en l'adaptant à vos usages et à ceux de votre milieu professionnel, nous tenons à ce que vous vous conformiez aux éléments de syntaxe donnés ici – nous allons donc agir comme dans le début de l'apprentissage d'une langue en sanctionnant les erreurs de forme, afin que vous maîtrisiez parfaitement les bases.

1.6 Les bases du diagramme de classes

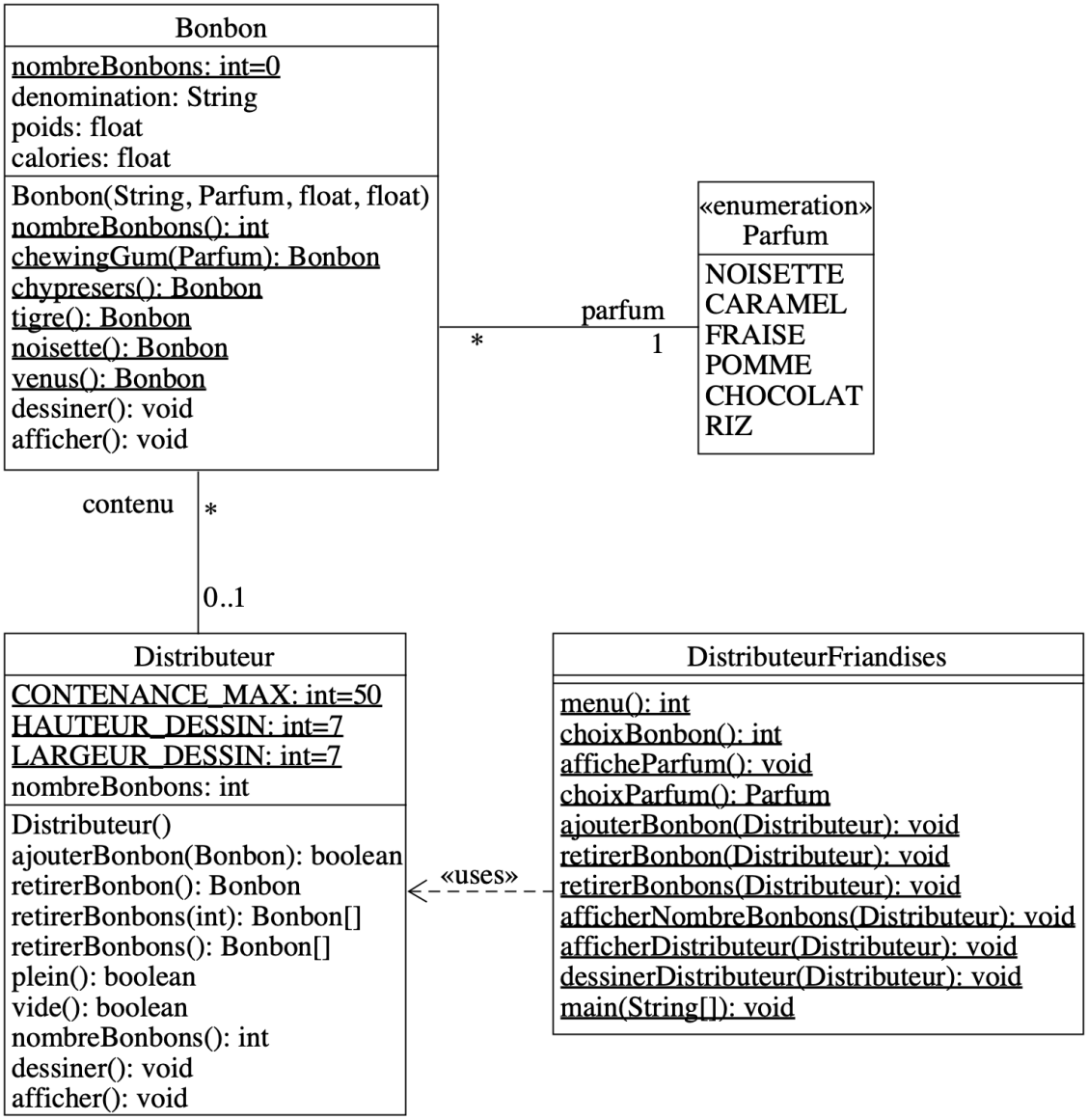
Le premier vocable UML étudié est le *diagramme de classes* : son but est de représenter des classes composant un programme (ou une partie de ce programme), leur contenu (partiellement ou totalement), c'est-à-dire les opérations (méthodes) et données (attributs) définis dans chaque classe (d'instance ou de classe), et surtout les relations entre elles : le plus grand apport d'un diagramme de classes clair est de permettre de visualiser très rapidement ces relations, et donc quelle classe utilise quelle autre classe.

Une classe est représentée par un rectangle détaillant :
— son nom,
— la liste de ses attributs,
— la liste de ses méthodes.

Suivant ce que le diagramme doit représenter, certains de ces éléments peuvent être omis (une partie ou tous les attributs, une partie ou toutes les méthodes) ; la représentation la plus simplifiée d'une classe est un rectangle contenant son nom.

Dans le premier TP de M1103 pleinement consacré aux classes et à la programmation par objets, les notions de méthodes et d'attributs de classe et d'instance ont été illustrées au travers de classes *Bonbon* et *Distributeur*.

Voici le diagramme de classes correspondant au TP *Distributeur de friandises* :



Le type énuméré `Parfum` (en Java, c'est une classe particulière) et la classe principale `DistributeurFriandise`, qui n'a aucune instance (elle n'a aucun attribut et ne dispose que de méthodes de classes) sont à part. Les attributs sont notés avec leur nom et leur type (après le caractère deux-points : `:`). Les méthodes comprennent leur nom et la liste des types de leurs paramètres entre parenthèses (les noms des paramètres peuvent être placés dans le diagramme de classes, mais sont souvent omis, comme ici, pour faciliter la lecture), suivi du type de valeur de retour (`void` pour les actions). Les méthodes et attributs de classe (`static`) sont soulignés (le reste sont des méthodes et attributs d'instance). Les constantes sont écrites en majuscule (comme en Java) et les attributs peuvent disposer d'une valeur initiale signalée avec = (après leur type).

Le plus important concerne les relations entre les classes. Un trait simple, continu, dénote une *association entre classes*. Une association correspond, en code Java, a un attribut (d'instance) d'une classe, dont le type est une autre classe. **Attention !** Les associations n'apparaissent jamais dans la liste des attributs d'une classe dans le diagramme de classes, mais sont clairement notées par ce lien : l'intérêt du diagramme est de pouvoir visualiser immédiatement ces associations. Les associations sont décorées par des *multiplicités*, obligatoires : le caractère `*` dénote *autant qu'on le souhaite*, un entier précis *n* (souvent 1), *exactement n*, et `i..j` un intervalle possible entre *i* et *j* inclus.

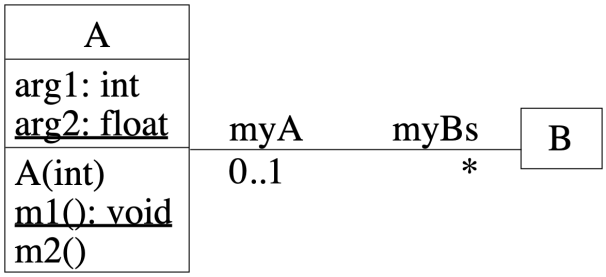
Contrairement à d'autres diagrammes, les multiplicités sont placées du côté de l'association qui est plus proche de l'objet qui a cette multiplicité (il y a un et un seul parfum par bonbon, dans ce diagramme). Les associations sont aussi décorés par des rôles (au moins un) permettant de caractériser l'association (c'est le nom de l'attribut correspondant dans le code Java).

On peut également avoir une relation d'*usage* entre classes, quand pour une classe A le code d'au moins une de ses méthodes fait appel à un objet d'une classe B, on note que *A fait usage de B*, avec un trait en pointillé terminé par une flèche, et décoré du stéréotype `uses`. Ici, la classe principale crée un objet de type *Distributeur* dans sa méthode principale `main`.

1.7 Récapitulatif

Les classes rendent des services au travers d'opérations : les méthodes, agissant sur des données pertinentes : les attributs, que ce soit de classe ou d'instance ; dans ce dernier cas, les instances représentent des entités différentes d'un même type, qui rendent les mêmes services. Le diagramme de classes permet de visualiser tout ou partie du fonctionnement des classes (opérations et données), ainsi que les relations entre ces classes (associations et utilisation).

Noter, sur le diagramme suivant : un *type de données*, un *attribut de classe*, un *attribut d'instance*, une *méthode de classe*, un *constructeur*, un *nom de classe*, une *association*, une *multiplicité*, un *rôle*.



2 Cours : Visibilité

Les attributs d'un objet sont normalement *les données qui lui permettent d'être fonctionnel* : un objet *encapsule* données et traitement afin de produire lesdits traitements. La finalité est de pouvoir *utiliser certaines méthodes depuis l'extérieur de l'objet*.

Pour faciliter la séparation de *ce qui est nécessaire au fonctionnement de l'objet*, différentes **visibilités** sont appliquées. La visibilité d'une méthode ou d'un attribut définit sa portée, et donc le fait d'être utile pour *soi* ou pour *les autres*. Java comme UML définissent quatre niveaux de visibilité :

Visibilité	Mot-clé Java	Symbole UML	Explication
Paquet	(aucun)	~	Visibilité par défaut en java, l'attribut ou la méthode est accessible depuis tout le *package*. L'objectif est de partager les données et opérations au sein d'un petit ensemble cohérent de classes.
Publique	`public`	+	Publicité totale : élément fourni à l'extérieur, en général service fourni par la classe sous la forme d'une méthode publique. Par exemple, la méthode `main` doit être publique pour servir de *point d'entrée* du programme.
Privée	`private`	−	Portée privée : cet élément est interne à la classe, seules les méthodes de cette classe peuvent y faire appel. En règle générale, les attributs sont privés (un accès en lecture peut parfois y être donné par des méthodes appelées *accesseurs* et conventionnellement préfixées par `get`, mais ce n'est pas toujours souhaitable). Certaines méthodes et certains constructeurs peuvent également être privés.
Protégée	`protected`	#	Utilisée dans des circonstances spécifiques qui seront détaillées plus loin.

3 Exercice – Transcription de code Java

Dans ce cours, nous étudierons souvent du code Java *sans détailler le corps des méthodes et constructeurs* (ce travail relève de la programmation). Voici la déclaration d'une classe *Boisson*, sans donner donc le code. **Exercice** : donnez le diagramme de classes (une seule classe ici) correspondant, avec types et visibilités.

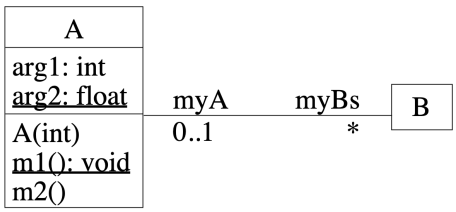
```
/**
 * Classe permettant de créer des boissons à stocker dans un distributeur
 */
public class Boisson {
    public static final String UNITE_VOLUME ="mL";
    private String denomination;
    private int volume;
    private double  kcal;
    private boolean gazeuse;
    private String gout;
    /**
     * Constructeur paramétré
     * @param maDenomination le nom commercial de la boisson
     * @param monVolume le volume de la bouteille en mL
     * @param monEnergie l'énergie totale donnée en kilo-calories
     * @param estGazeuse le fait de contenir ou non de l'acide carbonique
     * @param monGout une description du goût de la boisson
     */
    public Boisson(String maDenomination, int monVolume, double monEnergie,
                   boolean estGazeuse, String monGout) {
    }
    /**
     * Méthode permettant de connaître le nombre de kilo-calories aux cent mL
     * @return l'énergie /100mL
     */
    public double energieParCentMl() {
    }
    /**
     * Méthode permettant de savoir si la boisson est gazeuse ou non
     * @return un accesseur sur "gazeuse"
     */
    public boolean getGaz() {
    }
    /**
     * Méthode de fabrication permettant de créer un soda classique
     * (fait appel au constructeur de la classe)
     * @param nomSoda le nom commercial de la boisson
     * @param leGout une description du goût du soda
     * @return un soda, nouvelle instance de la classe
     */
    public static Boisson makeSoda(String nomSoda, String leGout) {
    }
}
```

4 Cours : Associations et Conteneurs

4.1 Rappel : association vers plusieurs objets

Pour rappel, les associations **doivent** apparaître dans le diagramme de classes comme des liens et non dans la liste des attributs, même si, dans le code Java, il s'agit bien d'attributs faisant référence à une autre classe ; la représentation claire des relations entre classes reste le but du diagramme de classes. Pour des associations dans lesquelles les multiplicités sont `1` ou `0..1`, le code est simple : il s'agit d'une référence à un objet, qui peut être nulle ou non. Mais dans le cas de multiplicités supérieures à un ou libres comme `*`, c'est différent.

Dans le cas d'une association entre classes comme dans le diagramme vu précédemment :



La classe *A* fait référence à *une collection* d'objets de type *B* (appelée *myBs*). En Java, on peut utiliser un tableau (le code Java de la classe A contiendra alors un attribut `B[] myBs`), c'est ce qui a été utilisé par exemple dans le TP *distributeur de friandises* (chaque distributeur dispose d'un tableau de bonbons). Mais les tableaux ont de fortes limitations, et dans les langages récents, comme Java (depuis longtemps), on préfèrera utiliser des collections ou, parfois, des tableaux associatifs : les **conteneurs**.

4.2 Les collections

Une *collection* est un objet représentant plusieurs objets, en général d'un même type. Les tableaux sont des collections. En Java, on peut également utiliser les listes (séquences) et les ensembles.

4.2.1 Les listes

Les listes en Java sont représentées par le type général `List`. Une de ses implémentations est `ArrayList` : les objets de type `ArrayList` se comportent comme des tableaux (on peut accéder immédiatement à un élément en connaissant son indice entre 0 et $n-1$ pour n éléments) avec des facilités de modification (la taille d'un `ArrayList` peut être modifiée à volonté, des objets ajoutés en début, en fin, au milieu, remplacés, supprimés...). De plus, l'API de Java fournit de nombreuses méthodes sur ces objets assurant leur facilité d'utilisation et leur performance.

Voici un exemple court d'utilisation de cette classe :

```
ArrayList<Boisson> boissons = new ArrayList<>();
  // Type paramétré et inférence de types
boissons.add(Boisson.makeSoda("TuringSoda", "Cherry")); // Ajout en fin
boissons.add(Boisson.makeSoda("AdaSoda", "Cola")); // Ajout en fin
boissons.add(1, Boisson.makeSoda("KolmogorovSoda",
  "Bergamot")); // Ajout à l'indice spécifié
System.out.println(boissons.get(2)); // accès en lecture à l'indice spécifié
System.out.println(boissons.size()); // accès au nombre d'éléments
boissons.set(0, new Boisson("NoetherJuice",
  500, 450.0, false, "Lime")); // accès en écriture à l'indice spécifié
```

4.2.2 Les ensembles

Les ensembles sont des collections sans indices, dans lequel chaque objet n'apparaît qu'une fois ; ce sont des structures utiles quand il n'y a pas de numérotation spécifique (du premier au dernier) pour des objets. En Java, le type général `Set` permet de représenter des ensembles. Deux implémentations sont fournies, `TreeSet` et `HashSet` : la structuration interne des données est différente mais le résultat est le même. Voici un exemple de lexique (ensemble de mots) :

```
TreeSet<String> lexicon = new TreeSet<>();
  // Type paramétré et inférence de types
lexicon.add("cat"); // Ajout
lexicon.add("dog"); // Ajout
lexicon.add("recursive"); // Ajout
lexicon.add("anamorphosis"); // Ajout
lexicon.remove("dog"); // Retrait - comparaison avec "equals"
if (lexicon.contains("cat") { // Test - comparaison avec "equals"
  System.out.println("Meow.");
}
for (String s : lexicon) { // Parcours
  System.out.println(s); // Ordre lexicographique ici
}
```

4.3 Les tableaux associatifs

Les tableaux associatifs ou dictionnaires permettent de représenter des associations plurielles : on peut les considérer comme des tableaux dans lesquels l'indice est un objet et non un entier entre 0 et $n-1$ (pour les tableaux de taille n). Les tableaux associatifs associent donc une *clé* (unique, correspondant à l'indice du tableau ou au mot du dictionnaire) à une *valeur* (l'élément stocké dans le tableau ou la définition dans le dictionnaire). Ce ne sont pas des collections, car ils ne portent pas sur un ensemble d'objets d'un seul type, mais de deux (celui des clés et celui des valeurs).

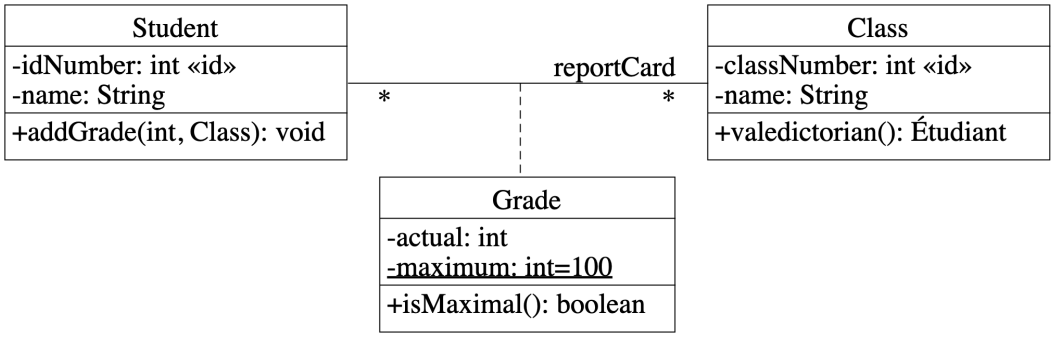
En Java, les tableaux associatifs sont représentés par le type général `Map`. Comme pour `Set`, deux implémentations principales sont fournies : `HashMap` et `TreeMap`. Voici un exemple de carnet d'adresses :

```
HashMap<String, String> addressBook = new HashMap<>();
  // Deux types - clé et valeur
addressBook.put("Mr. Johnson", "489, Elm Street");
  // Ajout : clé, valeur
addressBook.put(Ms. Marvel", "75, Sesame Street");
addressBook.put("Jane Doe", "999, Unknown Lane");
System.out.println(addressBook.get("Mr. Johnson"));
```

```
  // Accès par une clé
if (addressBook.containsKey("Ms. Marvel")) {
  // Test dans les clés - comparaison avec "equals"
  System.out.println("ok");
}
addressBook.replace("Jane Doe", "16, Ash Lane");
  // Remplacement d'une valeur associée à une clé existante
for (String name : addressBook.keySet) {
  // Accès à l'ensemble des clés (Set)
  System.out.format("%s : %s",
    name, addressBook.get(name));
 // Accès à chaque valeur par chaque clé
}
```

4.4 Les classes-association en UML

Une classe *A* contenant une référence vers un tableau associatif comprenant des clés de type *B* et des valeurs de type *C* peut dénoter une relation particulière entre les objets de ces trois classes : en UML, ce peut être une *classe-association* (d'autres structures de données peuvent aussi permettre de représenter ce type de relation). *A* et *B* sont liées de façon à ce que chaque objet de type *A* ait accès à plusieurs objets de type *B*, et chaque objet de type *B* ait accès à plusieurs objets de type *A*. Par exemple, dans un logiciel de gestion des notes, chaque étudiant est inscrit a plusieurs modules, et chaque module comporte plusieurs étudiants qui y sont inscrits. Mais pour chaque paire de *A* et de *B*, il existe un unique objet de type *C* caractérisant leur relation. Dans cet exemple, pour chaque étudiant (*student*) et chaque module (*class*), il existe une note (*grade*) que cet étudiant a obtenu dans ce module. En Java, on peut avoir dans une classe *Student* un attribut comme `HashMap<Class, Grade> reportCard;`. En UML, *Grade* est une *classe-association* notée comme ceci :



La classe-association est donc représentée sur l'association qu'elle caractérise, reliée par un trait en pointillés à cette dernière.

4.5 Diagrammes de classes et systèmes d'information

Les situations étudiées dans ce cours correspondent surtout à des *programmes* orientés objets, mais les diagrammes de classes peuvent aussi s'appliquer à des *systèmes d'information*, ou à des programmes s'appuyant sur des systèmes d'information et/ou des bases de données. Dans ce cas, représenter l'*identifiant* des objets peut être utile (cet élément correspond à la *clé primaire* d'une table). En UML, l'identifiant est repéré par le stéréotype *id*. Dans une classe-association, l'identifiant d'un objet est *le couple des identifiants des objets* auquel il est relié (la note du module 2104 pour l'étudiant d'idnum 777 est d'identifiant $(777, 2104)$).

5 *Exercice – Transcription de code Java*

Voici la déclaration de quelques classes Java, toujours sans donner le code des méthodes et constructeurs. **Exercice** : donnez le diagramme de classes (quatre dont une classe-association) correspondant, avec types et visibilités.

```
/**
 * A virtual person in a simulated environment
 */
public class Person {
    private String name;
    private String profile;
    private Avatar representation;
    /**
     * Standard parameterized constructor.
     * @param name An unique identifier
     * @param profile Can be empty
     * @param representation The avatar shown on screen
     */
    public Person(String name, String profile, Avatar representation) {}
    /**
     * Changes the public profile (set method).
     * @param newProfile The text of the profile to set
     */
    public void setProfile(String newProfile) {}
    /**
     * Connects to a new server.
     * @param server The URI to connect to.
     */
    public void connect(String server) {}
    /**
     * Attempts to interact with a virtual pet.
     * @param p The interaction target
     * @return True if successful (depends of the friendliness of the pet)
     */
    public boolean interact(Pet p) {}
}

/**
 * A virtual pet (AI)
 */
public class Pet {
    static Set<Pet> allPets;
    private String name;
    private String species;
    private Avatar representation;
    private Map<People, Affinity> friends;
    /**
     * Standard parameterized constructor.
     * @param name An unique identifier
     * @param species E. g. "cat", "dog", "ant"...
     * @param representation The avatar shown on screen
     */
    public Pet(String name, String species, Avatar representation) {}
    /**
     * Factory method used to create new pets periodically.
     * @return a new random Pet
     */
    public static Pet spawn() {}
    /**
     * Call this method periodically for random actions.
     */
    public  void doSomething() {}
    /**
     * Method called when people try to interact with this pet.
     * @param p the person trying to interact
     * @param true if successful, the person will be added as a friend
     *     or friendship will be upgraded
     */
    public boolean interaction(Person p) {}
}
```

```
/**
 * A class representing how a pet feels towards a particular person.
 */
public class Affinity {
    private Pet myPet;
    private Person myPerson;
    private int friendliness=1;
    /**
     * Parameterized constructor with a base friendliness level of 1.
     * @param thePet the pet in this relationship
     * @param thePerson the person in this relationship
     */
    public Affinity(Pet thePet, Person thePerson) {}
    /**
     * Method called when an existing relationship is improved.
     */
    public void increase() {}
    /**
     * Access a textual representation of the friendliness value.
     * @return text such as "friendly"
     */
    public String friendLevel() {}
}

/**
 * A representation for people and pets on screen.
 */
public class Avatar {
    private String image;
    private int[] location=new int[2];
    /**
     * Standard parameterized constructor
     * @param image The image to show on screen (file path)
     */
    public Avatar(String image) {}
    /**
     * Updates the screen with this avatar.
     */
    public void show() {}
    /**
     * Moves the avatar.
     * @param dx The horizontal movement (-1 to +1)
     * @param dy The vertical movement (-1 to +1)
     */
    public void move (int dx, int dy) {}
}
```